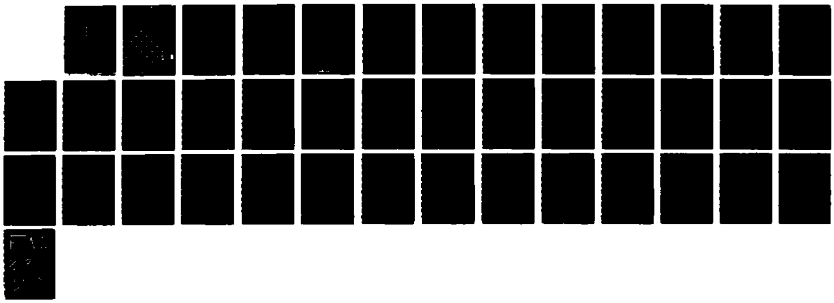
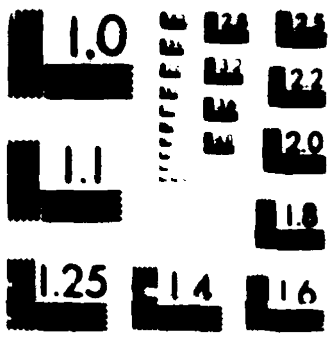
AD-A182 822 IDL (INTERFACE DESCRIPTION LANGUAGE): PAST EXPERIENCE AND NEW IDEAS (U) CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST J M NEWCOMER JUL 86 1/1

UNCLASSIFIED CMU/SEI-86-TM-9 ESD-TR-86-216 F/G 12/5 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

ESD-TR-86-216

(2)

Technical Report
SEI-86-TM-9

Carnegie Mellon University
Software Engineering Institute

IDL: Past Experience and New Ideas

Joseph M. Newcomer

July 1986

AD-A182022

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNLIMITED, UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT UNCLASSIFIED, UNLIMITED, DTIC, NTIS
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) SEI-86-TM-9	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-86-216
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. / 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARENGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JPO / 8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628 85 C 0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS.
11. TITLE (Include Security Classification) IDL: PAST EXPERIENCE AND NEW IDEAS	

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
63752F	N/A	N/A	N/A

12. PERSONAL AUTHOR(S)
JOSEPH M. NEWCOMER

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
FINAL	FROM ... TO ...	JULY 1986	38

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

THIS PAPER IS BASED ON THE AUTHOR'S EXPERIENCE IN CONSTRUCTING AN IMPLEMENTATION OF THE INTERFACE DESCRIPTION LANGUAGE (IDL). THE RESULT OF THIS EXPERIENCE WAS SOME INSIGHTS INTO LANGUAGE DESIGN, HUMAN INTERFACES, AND SYSTEM STRUCTURING, AS WELL AS METHODOLOGIES FOR THE COMPOSITION OF COMPLEX TOOLS. CERTAIN COMPLEXITIES OF THE IDL IMPLEMENTATION ARE DISCUSSED IN THIS PAPER, SHOWING THAT QUITE EFFICIENT IMPLEMENTATIONS ARE POSSIBLE. FINALLY, A SET OF INTERESTING DIRECTIONS FOR IDL AND IDL DERIVED SYSTEMS ARE SUGGESTED, INCLUDING PROGRAMMING ENVIRONMENT AND DATABASE RELATED WORK.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☒ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED, DTIC, NTIS
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630 / 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR   EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE

Technical Memorandum
SEI-86-TM-9
July 1986

IDL: Past Experience and New Ideas

by

Joseph M. Newcomer
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213

This work was sponsored by the Department of Defense.

IDL: Past Experience and New Ideas

Joseph M. Newcomer

ABSTRACT This paper is based on the author's experience in constructing an implementation of the Interface Description Language (IDL). The result of this experience was some insights into language design, human interfaces, and system structuring, as well as methodologies for the composition of complex tools. Certain complexities of the IDL implementation are discussed in this paper, showing that quite efficient implementations are possible. Finally, a set of interesting directions for IDL and IDL derived systems are suggested, including programming environment and database related work.

1 Introduction

IDL, the Interface Description Language, was developed at Carnegie-Mellon University as a second generation to the compiler research support system LG [10]. IDL has been used in the specification of Diana, the intermediate language for Ada™ compilers [6] [7], and was used as the core technology for the compiler automation tool set developed at Tartan Laboratories.

It is outside the scope of this paper to attempt to provide an in-depth discussion of IDL, either as a language or from a tutorial viewpoint. For these, the reader is referred to the IDL formal description [13] [14], the Diana description [6] [7] and the body of work from the University of North Carolina [18] [19]. This paper presents a brief introduction to the IDL notation, which should be sufficient for the topics discussed in this paper.

This paper consists of observations about and reflections on a particular implementation of IDL and, from this, speculations about the use and structure of IDL in the future. Some of these speculations are on the use of IDL in ways not thought about, or considered only vaguely, during its initial design. Most of these are extensions of the original IDL design goals, or implementation considerations which are in some way orthogonal to the high-level IDL design.

One of the interesting properties of IDL is that the semantic specification does not constrain the implementation strategies which may be used to achieve it. With the combination of the small design, formal model, and flexibility of implementation choices, I[1] consider it an interesting set of exercises to see how far the ideas can be carried. This paper is intended to suggest what I consider some promising directions for research based on the IDL model.

[1]In this paper, the first person singular and first person plural forms are not interchangeable. "I", "my", and similar forms refer to ideas of the author or more frequently opinions held by the author, and for which it would be unfair to distribute the blame. "We", "our", etc. refer to efforts which involved more people than the author, and for which it would be unfair to have the author apparently claim exclusive credit.

2 IDL as a tool

The IDL language is a specification language. When used in the context of a tooling effort, there is an associated IDL translator, which takes a specification written in the IDL language and converts it to a collection of specification and implementation files for a target language in which programs will be written. The simplistic model of an IDL runtime environment is shown below in Figure 1.

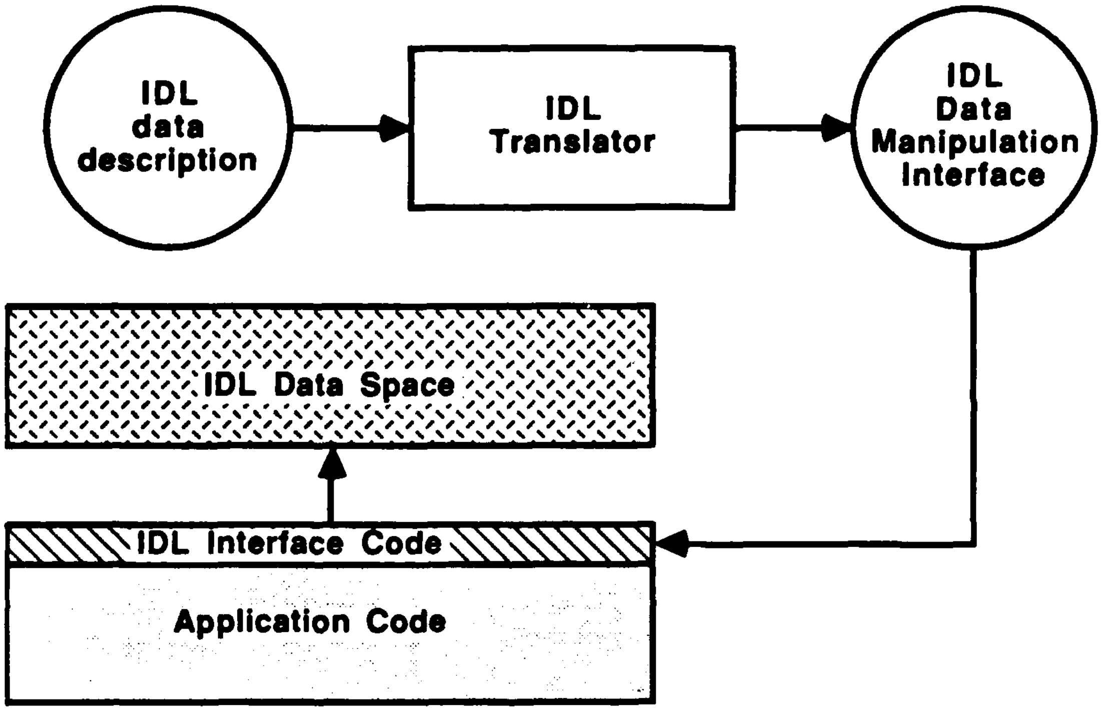


Figure 1: A simple model of IDL

The IDL data space consists of instances of data described by the IDL specification. It is created and manipulated by the user application code via an IDL interface. The task of an IDL translator and an associated IDL runtime support system is to provide the necessary interface between the user application code and the IDL data.

3 Philosophical Aside

IDL was one of the central tools chosen by Tartan Laboratories for its compiler construction tool set. One effect of using IDL at Tartan was that it formed the common domain of discourse for the Tartan tooling. The language was rich enough to support the basic data structuring required for applications tools such as symbol table generators, user interface generators, table packers, attribute grammar systems, and many other applications. Each application tool had an input language which used IDL as its core language and then included, in an extension language, application-specific specifications (for example, the symbol table generator allowed specification of the types of scope and nesting, whether overloading was present, etc.). The output from many of the tools was an enhanced IDL description which was fed to the IDL translator. Often such tools had associated runtime environment packages. For example, the symbol table generator system structure was as shown below in Figure 2.

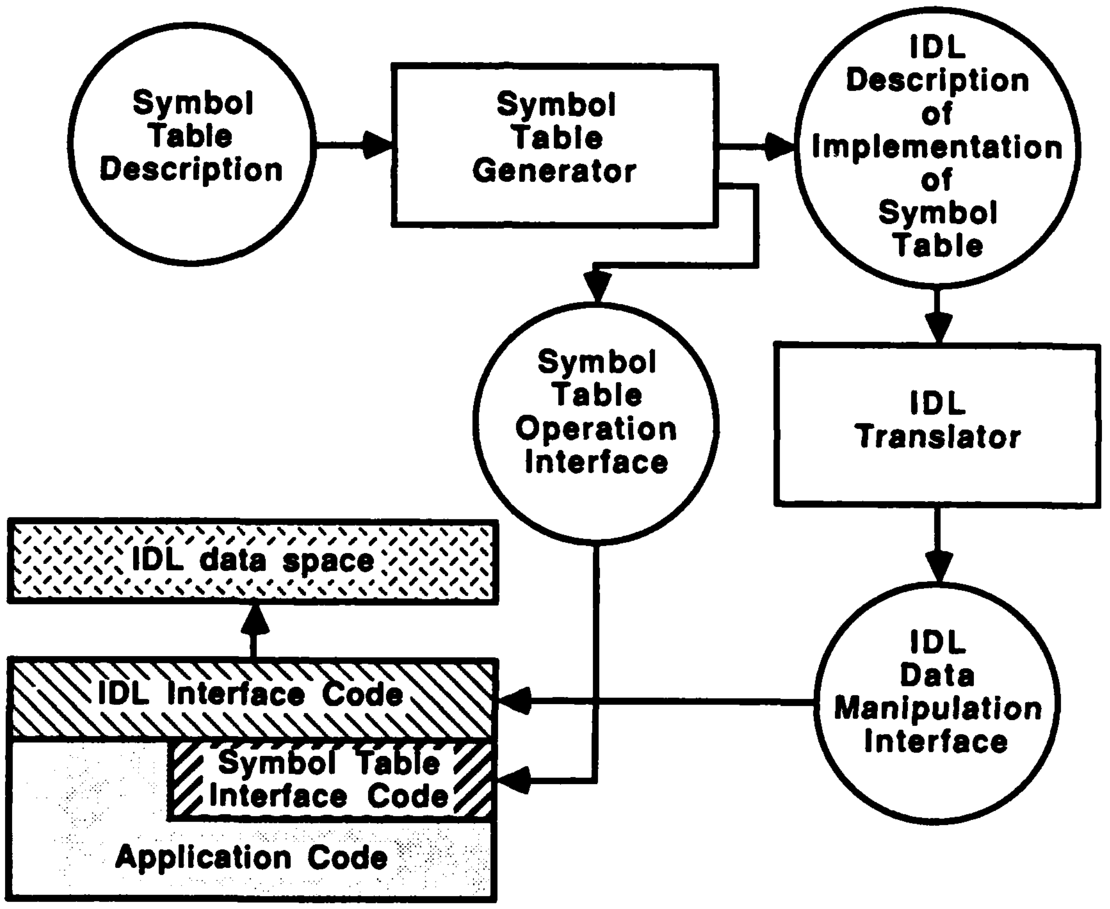

Symbol Table Description
Symbol Table Generator
IDL Description of Implementation of Symbol Table
Symbol Table Operation Interface
IDL Translator
IDL data space
IDL Interface Code
Symbol Table Interface Code
Application Code
IDL Data Manipulation Interface

Figure 2: Symbol table generator structure

As shown, IDL was not only a component of the input specification language to the tools (such as the symbol table description), but was also the output from the tool. Extended IDL descriptions supported a symbol table generator, table packer, user interface generator, and an attribute grammar system.

A direct benefit of this was that insofar as training there was only one data definition language to learn. From the implementors' viewpoint, tools such as the IDL translator and the table packer provided a "common object format" for many application tools. Because these provided machine-independent specifications, many tooling efforts did not require any machine-dependent knowledge in the tool; at the worst, representation specifications in the IDL definitions handled the few cases where such control was required.

I have therefore developed a strong bias in favor of the use of machine-independent languages (even those as low-level as Ada, C, Modula-2, Pascal, etc.) as the *output* of machine-independent tool sets, and uniform high-level notations (such as IDL) as input to these tool sets. (I am certain my categorization of the preceding languages as "low level" may seem strange, but read on!) As a simplifying piece of notation, I shall use the phrase "conventional programming languages" to include languages comparable to Pascal, C or Modula-2.

4 IDL Type Model

The IDL type model is significantly more expressive than the type models of most programming languages. The notion of non-hierarchical classes also gives it power and generality beyond Simula-67. This section will discuss the impact this more powerful model has on the structure of programs, and will then discuss briefly how one can integrate an IDL type system into an existing programming language.

4.1 Target Language Considerations: Objects

The basic aggregate data object in IDL is the *node*, which implements a concept similar to that of *record* in most languages. A node possesses *attributes*, analogous to record fields. These attributes are typed, and are references to *objects*. An *object* may have a simple scalar type (e.g., integer), a node type, or an aggregate type (set of or sequence of objects of some type).

There are many models for implementing the concept of a node, but the most common implementation is heap-based. This choice is often dictated by a desire to avoid various pointer-into-the-stack anomalies, or to accommodate languages (such as Ada) which have strong mechanisms to avoid this type of problem. However, it is important to understand that this choice is not *required* by the IDL definition; it is an implementor's choice but may be constrained by requirements of the target language.

4.2 Specification Level: Classes

Sets of node types which conceptually share information may be combined as a *class* type. A class type may specify attributes which are common to all objects in the class. However, a class is significantly different than the Simula-67 class; a class is not an object, and cannot be created; it is a name for a set of nodes[2]. (In the expected way, a class definition may name other classes, but this ultimately names a set of nodes).

An attribute is associated with a node or class by a production of the form:

node_or_class_name => *attribute_name* : *type;*

and a class is defined by productions of the form:

class_name ::= *node_or_class_name* | *node_or_class_name* | ... ;

where, as indicated, a class may be defined in terms of nodes or other classes. The attribute-defining productions make no syntactic distinction between assigning attributes to nodes or attributes to classes.

Taking a trivial example, and ignoring some of the subtleties of IDL, a description of the following form might be written:

```
N => truth: boolean;
N => count: integer;
N => subnodes: Seq Of N;
N => parent: N;
```

where a realization in some application language[3] might be

```
type N is record
      boolean truth;
      int count;
      array[0:?] of N subnodes;
      ref N parent;
end record;
```

A simple tree could be defined by the productions

[2]The semantics are more complex than this; this will be discussed in more detail later in the paper.

[3]Examples of application languages will attempt to convey intuitions of targets rather than reflect the precise syntax of any particular programming language, unless otherwise specified.

```
tree ::= inner | leaf;
inner ::= unary | binary;

unary => operand: tree;
binary => left: tree,
          right: tree;

leaf => value: value;

value ::= integer | name;

integer => value: integer;
name => name: string;
```

In this example, the attributes are declared only at the level of the nodes, i.e. the left side of an attribute production ("=>") does not appear on the left hand side of any class production ("::="). Classes are used to specify the types of the attributes. This example also illustrates that the names for nodes and classes and the names for attributes are in separate name spaces, and there is no conflict in assigning the same attribute name to different nodes (note the use of the word "value" as an attribute name in two separate productions and as a class name).

Attributes can be associated with classes as well; for example,

```
tree => depth: integer;
```

declares all members of the class `tree` to have an attribute `depth` whose type is `integer`.

It is worth pointing out here that in the original IDL publications a class was seen as a purely syntactic device for abbreviation, e.g., instead of declaring attributes individually on each node type, they could be declared for a class containing those nodes. Under this previous interpretation, it would have been the case that the declaration

```
inner => depth: integer;
leaf  => depth: integer;
```

would be identical to the declaration

```
tree => depth: integer;
```

Use of IDL has demonstrated that in fact a class has semantic significance beyond the simple syntactic abbreviation mechanism, both in the IDL conceptual model and in the resulting implementation model. Existing IDL documents [13] [14] still refer to the class mechanism as an abbreviation mechanism; a proposed revision 3 design (which includes a new formal semantic definition) is intended to reformulate the class as a semantic entity. The impact of classes on the implementation model will be discussed in section E. The implications on the target language model are that if the declaration is written

```
tree => depth: integer;
```

it is meaningful to have a program fragment of the form:

```
    var T: tree;
    ...
      T.depth := T.depth + 1;
```

because the type `tree` possesses a `depth` attribute, whereas if the attributes had been declared as

```
    inner => depth: integer;
    leaf  => depth: integer;
```

then the fragment would not be valid since the type `tree` does not have a `depth` attribute, even though any particular instance of the type tree (`inner` or `leaf`) does.

4.3 Target Language Considerations: Classes

At the user's conceptual model level, the class can be used as a mechanism for effecting generic procedures, a particularly useful mechanism in languages which do not possess such a capability. A procedure which operates on a class may operate directly on any attributes defined in the class, regardless of the actual node which may be operated upon. Thus, many of the questions generated by the presence of true generic procedures in a language (such as optimizations, collapsing common code bodies, etc.) can be totally avoided.

As an example, consider the a language with generic procedures; one might wish to write a procedure to increment the `depth` attribute:

```
    generic(T) procedure inc( N: T)
        N.depth := N.depth + 1;
    end inc;
```

The same effect could be obtained in a language with overloading, but without generic procedures, by the following:

```
    procedure inc(N:inner)
        N.depth := N.depth + 1;
    end inc;

    procedure inc(N:leaf)
        N.depth := N.depth + 1;
    end inc;
```

The same effect can be can be obtained by declaring a non-overloaded, non-generic procedure of the form

```
    procedure inc( N:tree)
        N.depth := N.depth + 1;
    end inc;
```

Such procedures are obvious and natural in languages such as Simula-67 but not possible in conventional programming languages without extremely careful use of features such as union modes or variant records. The complexity of achieving this for complex class hierarchies, or the even more complex case of non-hierarchical classes, is frequently intimidating and often unmanageable and unmaintainable.

I refer to procedures which can accept class types as parameters but which require neither overload resolution nor generic mechanisms in order to be realized as "imitation generic" procedures. They are, in general, much less powerful than full language-supported generic mechanisms and suffer some limitations over full language-supported overload mechanisms, but are a substantial improvement over the simplistic mechanisms supported by conventional programming languages.

An interesting problem with respect to both overloading and generic procedures in languages which support them is the "documentation problem": for what types is this procedure name overloaded or for what types may this generic be instantiated? Some languages which support polymorphic procedures (such as Smalltalk, if one may loosely apply the term "procedures" to the operation invocation mechanism used in that system) provide extensive and comprehensive support for the user in the form of "browsers". Modern programming environments being constructed for Ada provide similar mechanisms.

When an IDL class is used as a parameter, the range of types which are valid parameter types to the procedure may be determined by knowing only the parameter type and the class definition. It is certainly clear that there are advantages to knowing the scope of the types for which an instantiation is valid, but this requires that the set of types can only be extended by adding new members to the IDL class definition. This severely limits the generality of such "generic" procedures, a feature which may or may not be desirable.

When IDL classes are used to make procedures polymorphic, a mechanism as complicated as generic instantiation is no longer required; the ordinary compilation process generates a single code body which is shared by all the members of the class (the "types" for which it is "instantiated"). This is quite similar to what the type mechanisms of Simula-67 or Smalltalk permit, where a procedure may operate on an object in a class but not know anything about the extensions to the class; but the non-hierarchical class model of IDL lends additional flexibility to how the procedures may specify the collection of object types on which they operate. Some of the questions of how to collapse multiple instantiations of generic procedures into a single procedure when the generated code bodies are identical also become moot; the ordinary compilation process creates but one body which is polymorphic on its input classes.

4.4 Information Hiding

In languages which support data abstraction, it is possible to specify a "visible" part of the implementation, which is exported to the clients of a package, and a "hidden" part of the implementation, which is private to the package providing the service. This simple dichotomy is not sufficient in many applications, and something akin to the view mechanism used in databases is more appropriate.

Consider the case of a multiphase compiler, consisting of phases P_1 through P_n. Information computed by a particular phase may, at the very least, be considered read-only by subsequent phases, or perhaps should be hidden entirely. If two phases, say P_1 and P_7, agree on a means of communicating information, it may well be desirable that P_7 be forbidden to modify the information and that P_2 through P_6 be forbidden even to examine it. Often this is done by a "handshake agreement" rather than a "contract agreement" between the initial coders involved, but over time and changes of personnel the (undocumented) limitations are lost and unanticipated dependencies are introduced. Even if documented, the limitations cannot be enforced by the target language[4]. If P_1 and P_7 renegotiate their agreement, they may suddenly find that P_5 now modifies the information and P_3 depends upon it, making the whole system refractory to change.

In the original LG system, a capability was provided but never exploited: the ability to selectively hide information at each phase of the processing. There were many reasons this was not exploited, not the least of which was the fact that the view had to be specified on a per-field basis for each node. Furthermore, we were unable to specify read/write restrictions because of the target language model used; a field was either visible to the programmer and fully accessible (and thus modifiable), or hidden and completely inaccessible.

A new design has been proposed and is currently under consideration for a revision of the notion of an IDL "process" specification [12]. It is impossible in the space available to discuss in detail what earlier IDL documents termed an IDL "process"; for this the reader is directed to [13] [14] and [18].

Briefly, an IDL process was intended to represent some activity on an IDL structure, which may include starting by reading in an existing structure and/or concluding by writing out a structure; during the process a structure or structures may be created, modified, enhanced, or even deleted. A set of invariants specify the input requirements (if any input were to be performed), output requirements (if any output were to be performed), and static structural requirements of the structure. The new proposal includes these capabilities, as well as being able to restrict selectively such operations as assignment to fields, reading from fields, whether or not new instances of particular nodes may be created, and other operation specifications.

A goal of this is that when properly used this mechanism enforces contractual agreements between phases on what operations may be performed on each node, class, attribute, or attribute value set. In our above example, phases P_2 through P_6 could not depend upon or modify the information agreed upon by phases P_1 and P_7, because such information would be both unknown (in principle) and inaccessible (in practice). An example of providing controlled views is shown below in Figure 3.

An interesting historical development is that initially IDL emphasized the data specification portion of the abstract type model; the new proposal is beginning to deal with the operation specification portion of the abstract type model.

[4]And who reads or believes documentation?

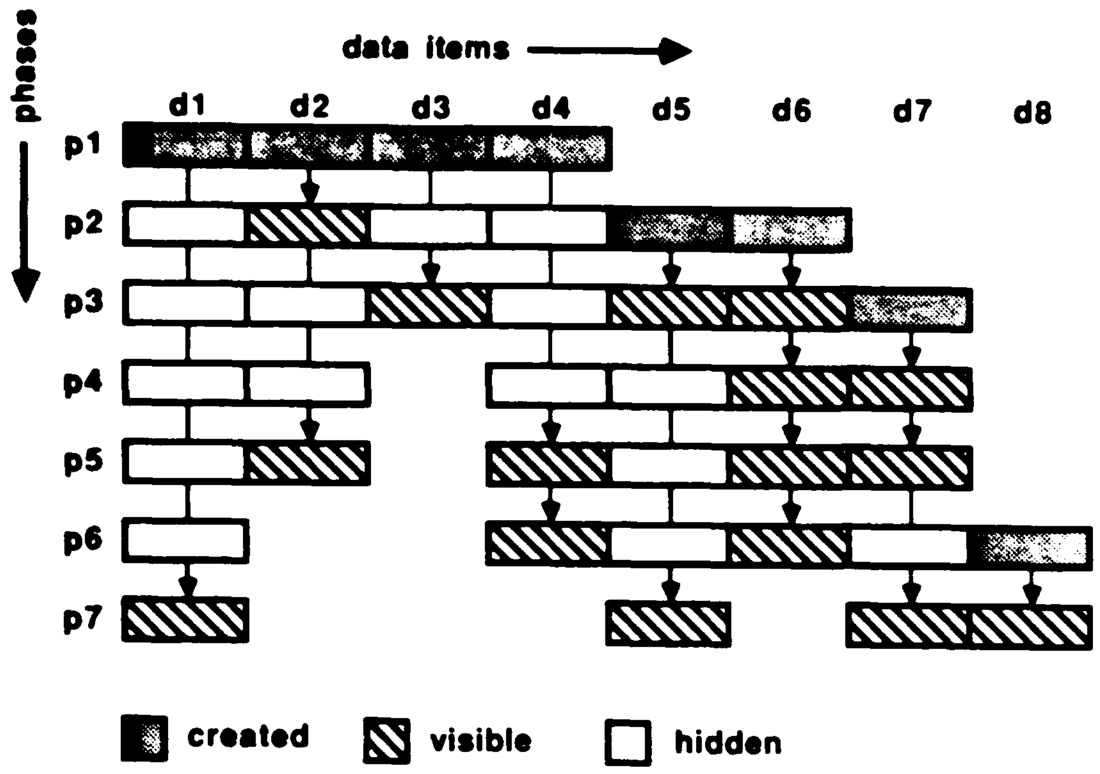


Figure 3: Controlled views

4.5 Target Language Considerations

The typical IDL implementation involves taking the IDL description, running it through a translator program, and obtaining as output some definition files which when combined with source files from the target language allow the user to write code using IDL objects. Typically these definition files will include type declarations, define record types in particular, provide enumeration literals, and of course define various operations that can be done to the data objects.

A significant problem with the power of IDL, however, is the fact that it is much more powerful than conventional programming language type systems. The result is that the programming language type system, in effect, fights the type system used by IDL. This is compounded in some languages because the language designers still foster the illusion that programs are written by human beings, and make no provision for the case where programs are written by other programs. Mechanisms of type safety introduced for the necessary and desirable goal of preventing "stupid" errors when human programmers are involved are not only excess baggage but require immense overhead to circumvent when programs are constructing code. Global disabling mechanisms are not sufficient, since it is often the case (such as with IDL) that human-written code and machine-written code are intertwined; one must be checked carefully while the other must not be. Even low-level languages such as C encounter significant problems [20].

Part of this is caused by a desire to maintain a language interface consistent with the target language. For Ada or Modula one wishes to use dot-qualified field selectors; for C, dereferencing arrows ("->"); etc. Operations such as assignment should remain as ":=" or "=". Furthermore, a goal which often introduces significant complication to the target language model is a desire to exploit the compile-time type checking mechanism of the target language to provide type checking for IDL data types.

Because of the power of the IDL type model, it is extremely difficult to meet all of the above requirements. In Ada, for example, the best we could come up with was an assignment procedure (ideally to be compiled in-line) so that the left and right sides could be properly type-checked at compile time. The use of **inout** and **out** parameters was impossible to properly type-check, because of the extremely strong type model of Ada. Various restrictions had to be placed on what could be written by the user in order to preserve type checking for IDL types at compile time.

As an example, since Ada forbids procedures from returning reference (access) types as parameters and forbids procedures on the left-hand side of assignments, and in addition does not allow the selector operator (dot) to be user defined, a number of rather cumbersome circumlocutions are required. Without supplying excruciating detail, consider a simple program fragment in an idealized language:

```
var N1: node1;
var N2: node2;
...
   N1.a.b := N2.x.y;
```

IDL does not require that `N1` or `N2` be implemented as record types, or be heap-allocated, or that the attributes be necessarily stored as bit patterns in memory. A read-only attribute may well be computed dynamically (writing such dynamic attributes is an amazingly complicated topic, and outside the scope of this discussion). However, the only way one can support data abstraction in Ada is to use a procedural interface, since dot-selection reflects implementation decisions which should not be visible to the user. Thus, at the very least the code would have to be written as

```
b(a(N1)) := y(x(N2));
```

as our expression. But since procedures cannot appear as left-hand-side values the code must be written as

```
store(b(a(N1)), y(x(N2)));
```

which is fairly unnatural (the use of overloading to have functions that evaluate to left-hand-side and right-hand-side values is also necessary, but a subtlety too deep to discuss here). In addition, if there were a procedure which took an **in** parameter:

```
proc print_value(in i: integer)
```

then it could be written as

```
        var T: tree;
        ...
        print_value(depth(T));
```

but if the procedure took an inout or out parameter

```
    proc increment(inout i: integer)
        i := i + 1;
    end increment;
```

then an attribute access which involved a procedural interface could not be used:

```
        var T: tree;
        ...
        increment(depth(T));   -- illegal!
```

although it could be written as

```
        var T: tree;
        var i: integer;

        ...
        i := depth(T);
        increment(i);
        store(depth(T),i);
```

In general, this set of limitations is not acceptable. Users do not wish to distort their thinking to accommodate incidental restrictions; remembering completely arbitrary limitations and circumlocutions is painful at best.

A major deficiency of most of conventional programming languages, including Ada, is that the abstract interface and the implementation are somewhat hopelessly intermixed. A typical example is the issue of how to represent an array of structured data: is it optimum to represent it as an array of records or as a record of arrays? In many cases the optimum representation for conceptualization is as an array of records, but the optimum representation for implementation (whether access speed or packing density) is as a record of arrays. Conceptually, the user wishes to write an access to some field in the i^{th} array element as

```
    Array[i].fieldname
```

but since the representation choice becomes explicit in the language it may be necessary (in order to achieve the desired space/time performance) to write:

```
    Array.fieldname[i].
```

If it is discovered well after the implementation has begun that the second representational form is required, or during a port that because of the target architecture that the second representational form is significantly faster on the new architecture, potentially massive amounts of code must be changed. There is no easy way to indicate that the mapping of "Array[i].fieldname" is an abstract mapping because abstraction exists only at the procedural level, not at the subscript or selector level. The only solution is to use a procedural interface to effect the mapping, which has many other undesirable effects (see section 4). Since IDL allows high-level specification of the abstract interface but includes low-level representation control, a simple one-line change in an

IDL specification (for example, changing an array-of-records representation to one of record-of-arrays) would necessitate massive changes in the target code. This violates a principle of parsimony: small changes should have small effects.

An additional constraint which introduces significant complexity to the IDL target language interface is a belief that compile-time type checking is a good idea. If you subscribe to this belief (as I do), then it is desirable to have the target language compiler insure that illegal programs are diagnosed (insofar as possible) at compile time. In the presence of non-hierarchical classes the design of an interface to adequately support the compile-time type checking mechanism, the attribute access mechanism, and the parameter passing mechanism, and still maintain a style consistent with the target language interface for simple nodes and attribute values produces significant tensions. Several proprietary designs for Ada interfaces to IDL have been done; all required (in my opinion) significant design compromises to achieve the necessary goals. In general the user interface has been compromised to meet the functional goals, particularly compile-time type checking. Alas, since these are all proprietary, no citations to these clever designs can currently be made.

One solution which has been proposed is to use a pre-processor: in effect, to write in a language which is a superset of the target language in which the IDL type model is fully supported. The source language is then compiled into the target language, with all necessary idioms, restrictions, and circumlocutions handled by the preprocessor. Ordinary user code which uses none of the extended type model may be freely interspersed with the extended type model code. For most languages, the complexity of such a preprocessor approximates the complexity of a semantic analyzer for the language. With languages such as C, Pascal and Modula-2 this is not particularly complex; for Ada, an interesting and important target, the complexity is intimidating. Consider the conceptually trivial problem of deciding that the parameter to a procedure is **inout** and using a rewrite rule to create a temporary variable into which to store an attribute; this requires that the overload and generic instantiation methods used by the Ada compiler be invoked to determine exactly which procedure is involved so that its parameters may be examined and the appropriate decision made, which requires, in essence, a complete Ada semantic analyzer.

One of the more promising approaches is to simply design a language whose native type model is consistent with IDL, but which is otherwise in the Pascal/Modula-2 class of language complexity.[5] Such a language could then be compiled into a variety of target languages, such as C, Pascal, Modula-2, and Ada, in the latter case using only a small subset of the total power and complexity of the language. This course is not without its pitfalls (as is any language design project); in addition to the usual language design issues, the goal of generating high-level language output introduces the additional problems of compiler incompatibilities and idiosyncrasies, machine-dependent targeting problems, and of course the incompatibilities between the IDL type

[5] I realize this reflects a bias toward statically-type-checked procedural languages. This paper assumes that such languages are useful and desirable.

model and the limitations of the target language type model. The advantage of this is that the output language might contain idioms, phrases, and techniques which a user would consider horrible or which, if written by a user would result in unmaintainable code, but which are legitimate for an automated tool to write; in the same way a compiler is free to produce outrageously obtuse machine code, a translator from the IDL-oriented language to, say, C, is free to write outrageously obtuse C code.

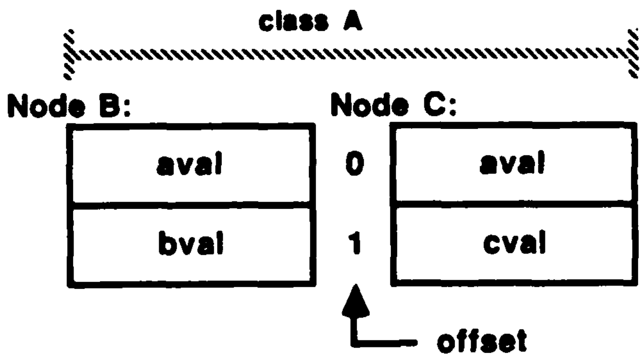
5 IDL/Target Language Packing Considerations

The designers of IDL had an implementation model in mind during the design process. An important consideration was the use of classes. If a procedure is to operate on a class, and access attributes of the class, it is clearly desirable that this be no more expensive than accessing attributes of a simple node. If it were necessary to determine which member of the class was being accessed in order to determine the offset into the node where an attribute could be found the time performance would be unacceptable.

The intent was that all attributes in a class would be packed in the individual nodes in the class in such a way that they were all at the same offset within the class. Thus an IDL definition fragment of the form

```
A ::= B | C;
A => aval: integer;
B => bval: integer;
C => cval: integer;
```

where *integer* is represented by a 32-bit word might result in a packing of the form:

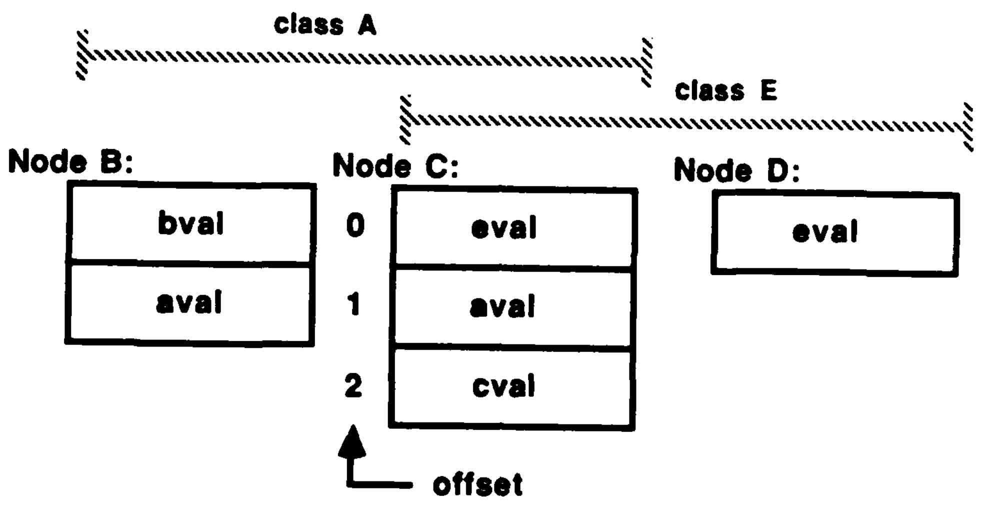


(It is worth observing that this is one of the many issues that emphasizes the fact that classes are more than simple syntactic abbreviations. Even in the early IDL implementations if `aval` had been declared separately in productions for B and C there was no constraint that it be of the same type in each or that it be packed to the same location).

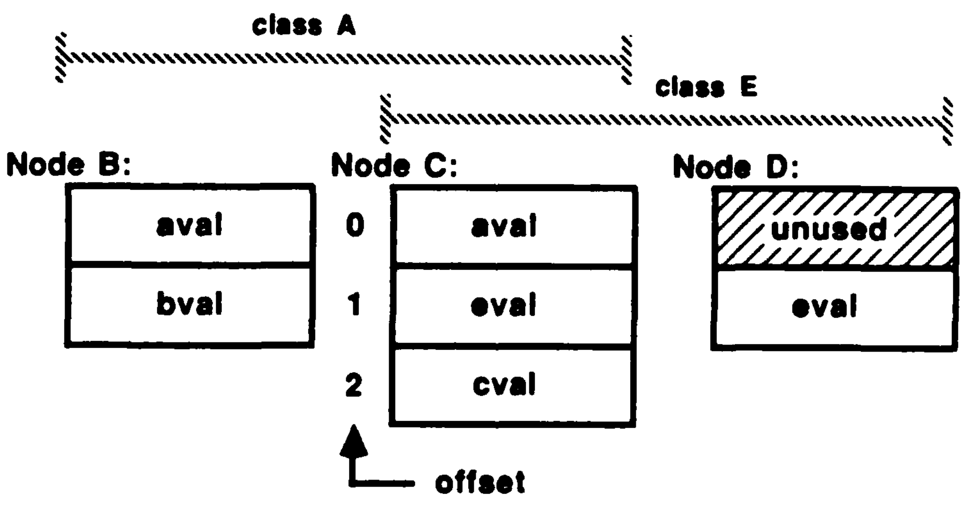
The complications of the packing requirement in the presence of overlapping or non-hierarchical classes should now be obvious; consider adding the fragment

```
E ::= C | D;
E => eval: integer;
```

to the above example. Now *one* of the possible packings of information is



while another (using a different packing strategy, such as "pack classes first") might be:



In this latter example, in order to satisfy the constraints of identical offset, some of the space in the D node is unused. It turns out that this space is often used by attributes unique to the node (or some class containing the node) for which such a packing is valid; so in practice the packing density actually can be very high.

While in general this problem is NP-hard, in practice simple heuristics and costing functions give very good approximations to an optimal packing.

6 Separately Compiled IDL Specifications

In the original IDL model, IDL is seen as being a single description encompassing all of a system. This model is not practical, as it makes it difficult to support separately compiled libraries in the traditional ways. The Tartan implementation supported separately compiled IDL definitions, but the mechanism was essentially a type-defeat mechanism at the IDL level which was checked only during compilation of the target language. There is a need for additional work in this area.

This problem with the IDL model became apparent after we began to use it at Tartan Laboratories. The assumption of a single global data specification ran counter to basic engineering requirements that separately compiled subsystems be able to be linked together without recompilation. Information hiding and modularity considerations required that client users not be able to modify the structure of information managed by various application packages such as parse tree generators and symbol table managers. Realities of computer time dictated that massive recompilation of every component of a system were infeasible (more on this later!) We therefore had to consider how to achieve, using IDL, what other systems with more conventional type systems already had: completely separate compilation.

In practice, we had been doing this for some time with IDL; IDL was originally targeted to the LG support system [10] which was part of the Bliss language [2] environment developed at CMU for the Production Quality Compiler Compiler project (PQCC). Since Bliss was an untyped language we could, and did, use this untyped property to allow various separately compiled IDL structures to reference each other. However, if a typed language had been used we would have had significant type violations. As we built our proprietary implementation language, Gnal, a modern strongly-typed language, the difficulties became apparent.

The solution was to define a base IDL type which had a single definition point shared by all IDL clients. This type was a generic type whose instances were the separately compiled IDL specifications[6]. Thus a specification, for example a symbol table, which needed to reference another specification, for example a name table, would simply name that type as an instance of the generic IDL type definition. This was accomplished in IDL by using a representation specification clause (one might argue about whether or not this is a "representation specification" or something else; indeed, a different mechanism is being considered in the draft 3 IDL specification):

[6]In Ada, this might be handled by using the derived type mechanism to create new derived IDL types from the base IDL type.

```
symbol => name: NTE;

type NTE;

for NTE use separate "name_table";
```

The use of separately compiled IDL descriptions also pointed out a serious limitation in many modern languages: the namespace problem. A node of type "A" in package "X" must be distinguished from a node of type "A" in package "Y". Unless the language itself provides unique-qualification mechanisms to disambiguate such names, the disambiguation must be done by a naming convention. While the desired goal might be illustrated as

```
use X;
use Y;

var A1: X.A;
var A2: Y.A;
```

it may be achieved in a language like C only by naming conventions, such as:

```
#include <X.h>
#include <Y.h>

X_A A1;
Y_A A2;
```

The conventions required for such subterfuges may require additional "representation" specifications to the IDL translator.

The mechanism used was a type-defeat mechanism; the "separate" declaration essentially said to the IDL translator "there is an external specification of this; trust me, and use this name to refer to it". Ultimately this was checked by the target language compiler, but it would have been much better if the IDL translator had checked all of this at the time it ran. Such checking requires that the translator be able to read interface specification files from other IDL runs. The use of abstract IDL descriptions (or "IDL intermediate languages"), such as Candle [4], are an important component of such a design. Candle and its implications are discussed in section G.

7 IDL-based tooling

The Tartan tooling suite encompassed many IDL-based tools. Each of these tools read an input language which included IDL as a subset. A uniform extension mechanism was added for supporting the appropriate tool-specific declarations.

In retrospect, I consider this to have been a good abstract decision which had significant implementation problems. For example, it meant that each tool had to parse and to some degree semantically analyze the IDL language, and in addition most had to reconstruct syntactically correct IDL source as its output. Even when these tools shared code for these purposes (as some did) it created unnecessary overhead in the tool development.

The Candle description CANDLE has been developed at the University of North Carolina. Candle is to IDL what Diana is to Ada: an intermediate representation of a source program. Candle, like Diana, provides both a syntactic structure and a semantic structure. Thus the burden of parsing and analyzing an IDL description is handled where it properly should be: in an IDL front end. Candle is a communication specification from an IDL front end to various application back ends, such as target language interface generators (thus forming what is now thought of as an IDL translator) and tooling generators.

IDL-based tools may now be written in a declarative style, but when an IDL description is required, the appropriate **with** clause naming an IDL compilation output of a Candle description would be given. For example, a symbol table generator input file might look something like

```
declare primary_symbol_table is
   with user_application.structure_name;
   symbols are ...;
   scopes are ...;
   ...
end primary_symbol_table;
```

where the output of this would be a new Candle structure which could be fed to an IDL target-language module.

In the same way that the designers of Diana imagined Diana representations of Ada source supporting code generators, flow analyzers, syntax-directed editors, cross-referencers, and dozens of other tools, the Candle designers expect this representation will support IDL-based tooling. It is being used for this purpose already in the University of North Carolina IDL Toolkit.

8 Programming Project Support

8.1 Recompilation Triggers

One major engineering problem in working with large systems is that small changes in central definition files of the system can result in truly major expenditures of computing time as the dependency ripples out through the system. This is particularly painful in any system in which the definition file includes more than simple procedure interface specifications. C header files which contain macro specifications are prime examples; however, any language in which an in-line procedure can be defined will trigger such events.

In order to rebuild a system after some specification file has changed, some number of modules must be recompiled. The worst-case scenario is that all modules of the system are recompiled, whether they need to be or not. The best-case scenario is that only the modules which are affected by the change are recompiled.

The recompilation process could be considered as an application of a predicate to a module and the specifications it references. If M is a module and $S_1..S_n$ are the specification files it references, then a recompilation predicate R could be defined

$$R(M, S_1, \ldots, S_n)$$

which returns "true" if the module M needs to be recompiled and "false" if it does not need to be recompiled.

The degenerate case which guarantees consistency is the predicate which always returns "true" no matter what its arguments. This predicate is trivial to implement but incurs a high cost when a change occurs. A predicate which is frequently used is one which returns "true" in almost every case where recompilation is required; it is implemented by the system builder looking at the list of modules and for each one saying "I think that one requires recompilation..."; unfortunately, this is a somewhat flaky predicate. If the system is large, if the dependencies are complex, and/or if the person making the decision is not totally familiar with the system structure (eidetic memory is a great help in this task), some module which requires recompilation will not be recompiled. This scenario is familiar.

The simplest program which, if used properly, guarantees consistency is the Unix[7] 'make' facility. This takes a specification of the dependencies and, if any specification module has changed, forces a recompilation of all modules which depend on it. It has the significant drawback that the user is responsible for guaranteeing the consistency of the dependency graph described to the 'make' program and the actual dependency graph of the system being constructed. The 'make' facility predicate requires knowing the output, O of a "compilation", and thus applies a predicate

$R(M, O, S_1, \ldots, S_n)$

if the module M or any specification file S_i is newer (by date stamp comparison) than the output O, the recompilation is required.

Programs such as the Unix 'make' facility which rely on primitive date-comparison algorithms cannot detect changes in content as distinct from changes in form (e.g., adding a comment). Various distortions are required to circumvent major rebuilds, all of them requiring manual intervention and usually error-prone. Introducing auxiliary definition files (a frequent practice among C programmers), removing actual dependencies from the 'make' file, and utility programs which update file dates are all common work-arounds.

8.2 Recompilation Predicates

Work began at Tartan in 1980 to develop a more sophisticated predicate than a simple date comparison. In particular, adding upward-compatible specifications to a definition file should not trigger massive recompilations. (Independent work on this problem was done by Walter Tichy at Purdue [17] for the C language, and was reported by Evans *et al.* for Praxis [11]; many others have, or are now looking at this problem).

In modern languages with overloading, it is certainly true that adding a procedure could introduce an overload conflict, and failing to recompile all modules which use the definition would mean that the inconsistency would not become visible until the next major system build. However, in prac-

[7]Unix is a trademark of AT&T Bell Laboratories

tice the probability of such an event is actually rather low, and a more liberal interpretation which significantly reduced recompilation costs was chosen for the Tartan implementation. Under the liberal interpretation, any specification file which had changed but which was identical to or upward compatible with a previous version of the same file would not trigger a recompilation event for the files which depended upon it.

Consider the issues of upward compatibility if a procedural interface is used. If the input and output specifications are held constant the body can change without affecting the interface; the new interface is identical (and thus compatible) with the old interface. In a package definition, the addition of a new operation on the data types described in the package is (in general) an upward compatible change; existing code using the existing interface is not rendered obsolete, and new code which requires the new operation would be recompiled simply because it has been changed to use the new operation. Changing the number or type or mode of the parameters to a procedure, or the result type, would not be an upward compatible change; however, only the modules which use that particular procedure need to be recompiled, *not* all the modules which use that interface specification file. The specification is upward compatible for all clients except those that use the changed procedure.

However, when code is compiled inline, a new dependency is introduced. The validity of the code in client modules now depends not only on the abstract specification but also upon the implementation. The modification of the body of a `#define` in a C header file is an example of modifying the implementation while holding the interface constant. All clients of that definition must be recompiled.

The impact of IDL on the smart recompilation system used a Tartan, and which would be true for any other system supporting inline expansion of the IDL structure accesses, was such that it was very unlikely that an IDL target language module was upward compatible with a previous version. This is because the most frequent change was adding or deleting attribute definitions, and this almost always resulted in a repacking of the data structure, so that all of the field offsets changed. This meant that the inline bodies of the accessing operations changed; even though the interfaces remained constant (requiring no code changes) the implementation changed, and recompilation of all clients was necessary. The upward-compatible predicate did not (and could not) ask on a per-module basis if the module used any bodies which changed; if any repacking was done, all client modules had to be recompiled. Thus, any change in an IDL definition file resulted in the complete recompilation of all files which depended upon it. (In addition, the predicate did not detect "upward compatible" changes of the structure, although this certainly could have been added). Because the compiler also supported in-line expansion of user-defined functions (which could access IDL data objects), the recompilation effects could frequently encompass the bulk of the system.

8.3 Minimizing Repacking

When using conventional hardware architectures, the use of inline expansion of IDL accesses meant that any change in the IDL definition which resulted in a repacking would invalidate existing code, and there is in general no good way to determine if such repacking has occurred. Recompilation can be minimized only if a *compatible* packing of information can be maintained between separate executions of the IDL tool.

An important implementation feature which any IDL system should possess, then, is what I have called "cached allocation", an idea I first encountered in a slightly different guise in the SIL circuit-drawing package developed at Xerox PARC [15]. The particular aspect of SIL which was interesting to me dealt with how SIL handled the gate-packing problem. In many integrated circuit packages there are multiple instances of the same logical component, e.g., four 2-input NAND gates in a type 7400 chip. SIL would "pack" unallocated NAND gates into uncommitted packages. The packing was then recorded. On a subsequent run, if this assignment of gates and pinouts to physical packages was satisfactory then no repacking was done; the result was that the existing wires on the board could be left in place. Any new NAND gates added to the drawing would be packed into remaining uncommitted chips.

As adapted to the IDL environment, the basic idea of cached allocation is that the structure packer is normally run only *once* on a definition file, and that packing is recorded in a "cache". In subsequent runs, the previous packing is read in and used, and any new fields are "tacked on the end" of the data structure. Thus, all previously compiled modules will still have the correct field offsets for the existing fields; any modules that reference the new fields have obviously been changed and will, as a matter of course, be recompiled. Only when space utilization degenerates below acceptable levels (caused by class fields being added or deletion of existing fields) is it necessary to repack the structure. For example, consider the effects of adding a new declaration to the example given in section 5:

```
        A => nval: integer;
```

the optimum packing would be something like that shown below in Figure 4, but to use cached allocation and the packing shown earlier in section 5 which now must be preserved, it must be packed as shown in Figure 5 below, which artificially increases the size of the A node with some unused space. *If the next run adds a new declaration*

```
    E => mval: integer;
```

the effect becomes more obvious. At some point the wasted space will necessitate a new packing, thus invalidating all existing code and requiring a complete recompilation of the system.

A direct implication of this is that the node size may not appear in-line in any generated code; for example, using the C "sizeof" operation would require that any module which created nodes would have to be recompiled, as would the use of "new" in Pascal or Modula-2, since these operations typically are implemented as requiring compile-time knowledge of the size of the data object to be created. Unless the recompilation tool knows which modules perform "new" opera-

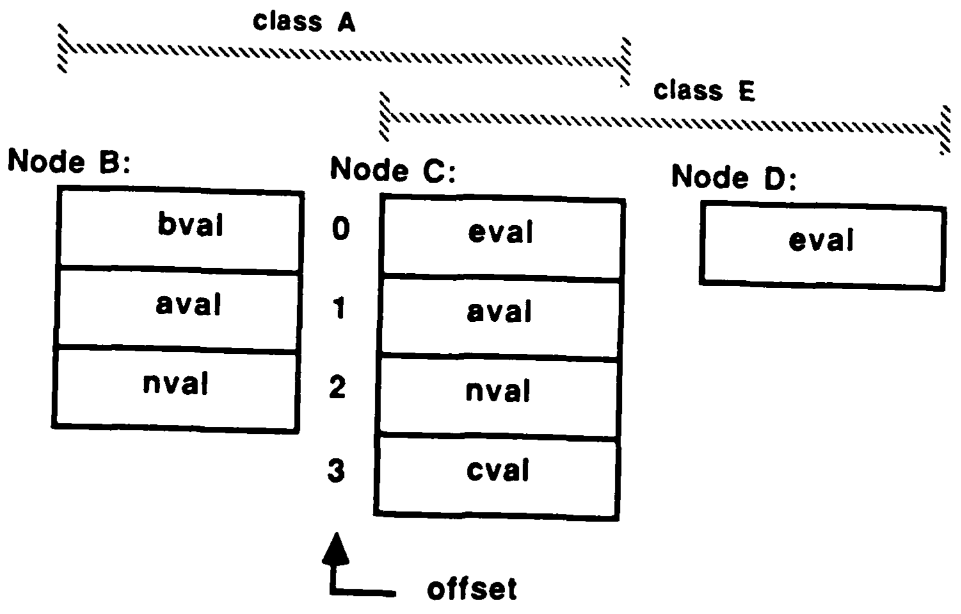


Figure 4: Optimum packing

tions, all clients of the IDL interface specification file would have to be recompiled if the node size changed.

However, if nodes are identified by unique numbers, and the sizes are obtained by a link-time binding of node ids to sizes, then there is no need to recompile any code which creates nodes; as the node size changes from run to run of the translator, existing code continues to run.

An important consideration in using this technique, however, is *monotonicity* of the alterations. A node may never be decreased in size. Since existing code is not recompiled, it can contain references to fixed offsets in the node which, if the node decreased in size, would now have undefined effects.

A very careful analysis of the effects of adding or deleting nodes, adding or deleting attributes, and adding or deleting nodes from class definitions is required to determine the effect of such changes on the validity of existing code. Careful engineering of the IDL target mapping and run-time environment in response to these changes is also required. Some of the simpler cases are discussed briefly here.

However, it is now possible to have the IDL tool interface to the target environment to support smart recompilation, even when the target environment cannot. By "lying" to the 'make' facility about the interface specification file date, the need to recompile modules can be avoided. Having a tool with complete knowledge effect the subterfuge is considerably more reliable than having a user with incomplete or possibly incorrect knowledge attempt the same trick.

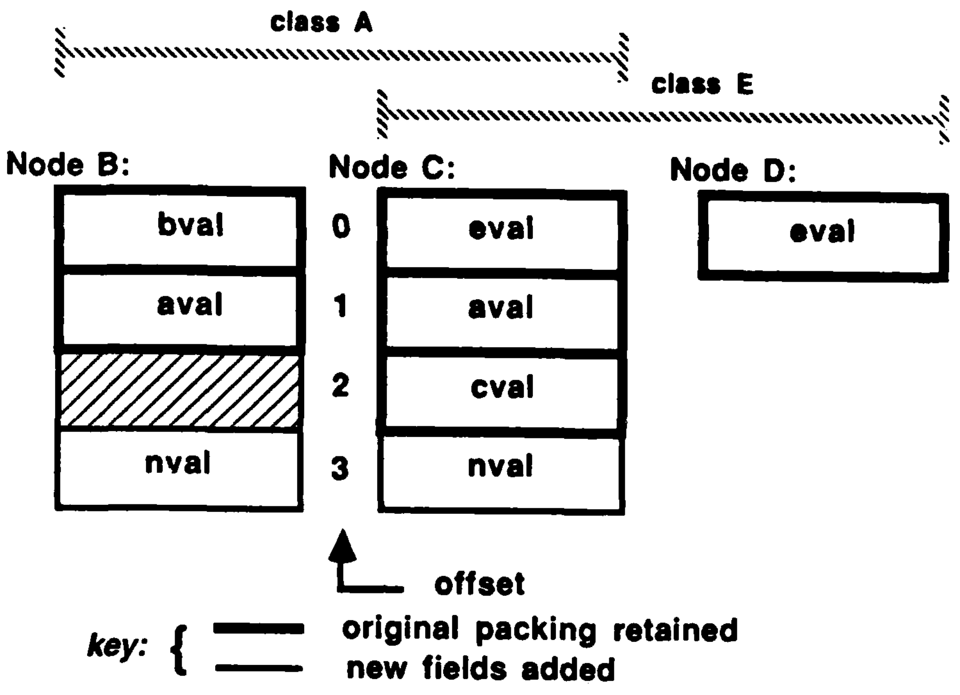


Figure 5: Cached allocation

When cached allocation is used, some careful analysis of the possible changes the user can make and their impact on compatibility must be made. In some cases, careful engineering decisions about the runtime environment can allow certain decisions to be bound sufficiently late in the process that changes can be upward compatible; the node-size example cited above is one such case.

Adding attributes is a simple problem when cached allocation is used. Such new attributes are, as shown above, appended to the end of the structure. If the attribute is an attribute of a class, all members of the class must be extended so that the attribute can be appended to the node of maximum size. This may create significant wasted space in the class members, and eventually a repacking will be required.

Deleting an attribute requires some consideration of the overall effect on the system development. One solution is to delete the attribute and the language interface to it, but otherwise assume the existing compiled code is valid. Such existing code (which may reference the attribute) will not be recompiled unless there has been some other triggering criterion (such as modification of the file), and any recompilation of such files would generate errors. Of course, existing uses of the attribute within the (un-recompiled) system may now find meaningless information in the (now unused) attribute, leading to erroneous program behavior. In the absence of more sophisticated smart recompilation strategies, such as those described by Tichy [17], the actual decision is more a managerial strategic decision than a technical decision; it is based upon considerations such as (alleged) knowledge of the usage and recompilation costs; the IDL translator should permit a high-risk option of minimum recompilation. Whether smart recompilation is used or not, the monotonicity must be preserved, the "unused" field may not be allocated for any other purpose because it is purportedly "free"; it is "undefined" but "unavailable" for future modifications.

Note that this above discussion suggests something which I said earlier was a bad idea: letting the user make the decision about whether to recompile a system or not. While I believe this as a principle, there are times when a simple cost/risk analysis shows that the risk is low and the cost is high, e.g., I just added an attribute, decided after a quick test that it was the wrong thing to do, and deleted it. If I must take risks, I would rather have a reasonably controlled expediency mechanism than a totally anarchic one.

Continuing on, nodes may be added to a class. Again, recompilation minimization depends upon the usage patterns and cost/risk analysis. Most procedures which operate on objects in the class will continue to operate even though a larger set of object types is permitted. Only those cases in which a complete discrimination of class members appears within the procedure body will there be any anomalies. Proper initial construction of the code (for example, not assuming a "case" discrimination on a class gives complete coverage and including appropriate "otherwise" clauses) will reduce the risk of such changes. Code used to check class membership of a type within a class must also be a link-time rather than a compile-time binding. The link-time binding of class membership checking was carefully engineered in the Tartan implementation, but it would take too long to describe here.

Obviously decisions which require human intervention are always subject to error. The high payoff of using cached allocation and high risk/low risk/no risk decision mechanisms are that it is possible to reduce the cost of change during the development/prototype cycle, particularly when rapid turnaround is required for productivity. Since all the decisions with risk involved have a no-risk fallback (to require the recompilation of all client modules), the no-risk fallback can be frequently handled by such strategies as off-time (usually overnight, or over-weekend) massive recompilations, while the higher risk strategies allow for rapid turnaround during the work-time (usually daytime) period.

8.4 Abstract Interfaces?

A typical argument against the complexity of the recompilation mechanisms and the careful engineering required to use them effectively is that the abstract interface "ought to" hide the implementation, so that accessing fields is done entirely through procedural interfaces which are insensitive to the actual "record" layout. Thus, the implementation can be readily changed without impacting the users at all.

This is certainly a laudable goal, but several realities interfere with it. First, procedural interfaces are rather clumsy to use; a separate "store" procedure must be used to modify each attribute and a "fetch" procedure to read one. This is somewhat unnatural, but given certain goals of the interface design and programming standards it might construably be an acceptable way to write programs. Obviously such linguistic features as user-definable selectors could eliminate this by providing the desired syntactic sugar.

The overwhelming argument against the procedural interface, however, is the access time argument. If the cost of a procedure call is nonzero (as it almost always is!), the performance of a system can be reduced by large factors, if not orders of magnitude, by having to use a procedure to read and write every attribute. Programmers, understanding such costs, then distort their code to fit it, caching attribute values in local variables and generally reducing the intelligibility of the code in an attempt to make it perform reasonably. Therefore, even if the procedural interface is a desirable programming paradigm, it is too expensive to use if implemented in a conventional language.

The next obvious step is to provide for in-line expansion of such a procedural interface. As soon as this is done, the generated object code now contains fixed offsets into the data structures, and is indistinguishable from code which used ordinary selectors (at least to someone reading the object code). The cached allocation model addresses the problems of generated target code, not the issues of abstract interfaces, and is, in fact, largely insensitive to the abstract interface (except for how to convince an environment that a new interface is "upward compatible" with the previous version).

Of course, all of this is an incidental property of our current computer architectures, in which the knowledge of how to access information resides in the code. If an architecture providing descriptor-based access to data structures were used, many of these problems simply disappear; knowledge of how to access the code would reside "with the data", or at the worst case in a single link-time or run-time bound definition. While I know of several sophisticated link-time-binding mechanisms for manipulating offsets, I know of very few, if any, linkers which can handle link-time data-size specification. In addition, the actual instruction (even if the size remains fixed) often changes depending upon the alignment of the data; bit-field extraction often requires multiple instructions so accessing an 8-bit byte-aligned field moved to a non-byte-aligned position is not something easily modified at link time unless one is willing to allow significant performance degradation.

Database systems responded to this problem years ago; most of them abstract the representation from the code ("representation independent code") but pay a high price in performance; certainly higher than would be acceptable for implementing data structures such as compiler data structures. In general, the only way sophisticated deferral of binding decisions can be implemented efficiently is if the compiler can be modified to support new kinds of inline access via strange implementation schemes (e.g., all data structure access via indirect descriptors). While sophisticated inline mechanisms sometimes help, they only succeed insofar as they can take advantage of existing code generation templates. If obscure instructions or strange addressing modes are required for maximum efficiency, and inline machine code insertions cannot be made, one is limited to what the compiler can support. The mechanisms described above could be implemented in most plain-vanilla-compiler environments.

All of the above complex mechanism is therefore in response to the reality of current architectures and how to cope with their all-too-real problems.

8.5 Implications on Target Language

To take full advantage of the packing techniques, both for ordinary IDL and for cached allocation, requires that the IDL system actually be able to control how a record is laid out in memory. The control may either be explicit (for example, the use of representation specification clauses in Ada) or implicit (the order of declaration of fields in C). For some languages it may be possible to subvert their type systems or use interesting substructuring to avoid the need for layout control for simple IDL, but it is not clear that such tricks can work across multiple compilers or when cached allocation is desired.

8.6 Implications on Target Environment

When interesting techniques such as cached allocation are used, the way in which the updated definition files produced by an IDL translator interact with the target *environment* must also be controlled by the IDL translator. As discussed earlier, in a simple C/'make' environment, to suppress unnecessary compilations it may only be necessary for the IDL translator to set the date of the new header file to the date of its previous version. For languages where the definition files are themselves compiled entities a more sophisticated mechanism will be required to defeat the environment's recompilation predicate. Ideally, a recompilation predicate should be general enough that the user could extend it; in practice, unified environments are typically "closed" to user extension and modification, being viewed as complete and sufficient. This is unfortunate, since the implementors of such environments cannot predict all possible uses, and even if they could would not have the resources to provide support for every conceivable use.

Implementors discovered that simple date-stamp comparisons can fail (either forcing unnecessary recompilations or failing to force a necessary recompilation), especially in distributed computing environments without centralized clocks, so in these environments more sophisticated mechanisms which include processor ids, compiler ids, and other information (such as checksums of the source files), some intrinsic and some incidental to the validity of the interface, are sometimes used. The increasing sophistication of such equality predicates is rarely based on the semantic content of the file, but is instead based on totally incidental and often unrelated properties, but which are "very fast" to compute. More sophisticated predicates based on actual semantic content are required [3] [17].

8.7 Lessons

There are some important issues here for language designers, compiler implementors, and environment implementors.

When programs are generating the record or structure definitions, bit-by-bit layout control may be required. If the language is sufficiently simple and the compiler is equally simple, the tool can take advantage of the compiler's data packing algorithm to indirectly effect control. Sophisticated compilers which give the user no layout control and which do "optimized" packing, however, are extremely difficult to interface to. While one may not wish to give layout control to a programmer, or document how it is done, such control and/or documentation is essential to program generating tools.

Mechanisms which are intended to prevent errors caused by failure to recompile dependent modules and which are based on anything other than actual semantic compatibility (e.g., changes in comments, names of formal parameters in interface specifications, indenting, and other incidentals must be ignored) at the interface level (and implementation compatibility when inline expansion is permitted) must have ways of being subverted by sophisticated tools which have precise knowledge of what constitutes compatible or incompatible changes. Better still, a well-defined notion of upward compatibility under both strict and liberal interpretations must be supported. If complex problems are ignored (such as upward compatibility of record structures under repacking minimization strategies in an IDL tool) the mechanisms will have to be subverted.

My basic complaint about programming languages and their environments was summed up rather succinctly by the statement "It is time we stopped designing languages in which people write code and started designing languages in which tools will write code" [8].

9 Data Base Interface

The nature of data is changing. Data now far outlives the programs which create it. The original goals of IDL were to specify a language- and machine-independent data description, to be able to communicate it in a language- and machine-independent fashion, and to be able to control its mapping to physical representations. This would allow programs written in a variety of languages on a variety of machines to communicate with each other, while still maintaining the efficiency required for internal manipulation.

To date, the realizations of this model have largely dealt with short-lived data, such as compiler interphase communication, representations of separately compiled module specifications, and similar applications. However, the descriptions of the data were encoded in the programs. In a database environment, the data description must live with the data.

In the Tartan IDL environment, there was long-lived data (such as the module interface or separate compilation files produced by the compilers) but the data descriptions lived in the producers and consumers, not in the files. This meant that any repacking of the information invalidated all of the existing interface files, because the formats were usually incompatible. Even the use of the proposed (but never implemented) cached allocation was not a sufficient solution, since the high-performance binary reader/writer modules would have to detect nodes stored in the files with one size but which had been extended to a new size.

To adequately use IDL in a database environment, the full IDL description of the data (often referred to in IDL documents as "the IDL symbol table", [4]) must be part of the information transmitted when a connection to an IDL information structure is "opened".

It is worth noting that the use of IDL to describe and interface to a database (in the sense of a hierarchical, network, or relational database) rather than the classical memory-resident data model usually used for IDL appears to be an issue of implementation engineering; the basic IDL specification mechanism appears to be quite suitable for expressing an interesting collection of database specifications. Several of us involved in IDL consider the use of an IDL model for databases, and comparison to existing data definition specifications [1], to be an interesting and promising area of research.

Another aspect of database system integration deals with separate IDL instances which must have cross-links. For example, consider the case where IDL is used by a compiler to represent *interface specifications*. There may be two modules of the following form:

```
module A
   use B;

   var bvar: BType; -- type BType is defined in module B
end A;

module B
   type BType is ...
end B;
```

The implementation strategy chosen (not necessarily the optimum one, but one which illustrates this example well) is to separately compile module B and then module A, but not include any of the content of the module B interface file in module A's interface file. Thus the interface file for module A contains, in some form, a *reference* to information in module B's interface file (Figure 6).

After having participated in two efforts to implement this type of system, I have become convinced that the problem is exactly that of doing a classical relational "join" operation. This model, having a sound theoretical underpinning and suitable mathematical description, has clarified my understanding of the problem and promoted it from the domain of an "implementation hack" to something which can be reasoned about and communicated much more readily.

Consider a relational tableau presentation: it might show how the variable becomes associated with its type definition by a simplistic picture of the form shown in Figure 7.

This is obviously a great simplification; the actual tableux used would be far more complex: for

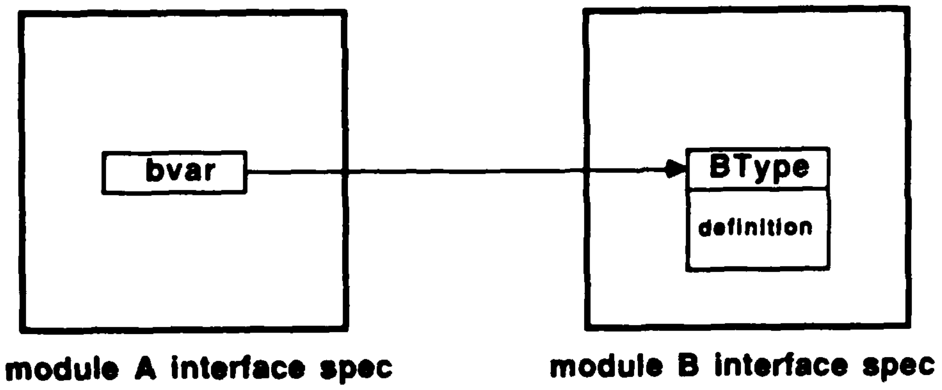


Figure 6: Reference between modules

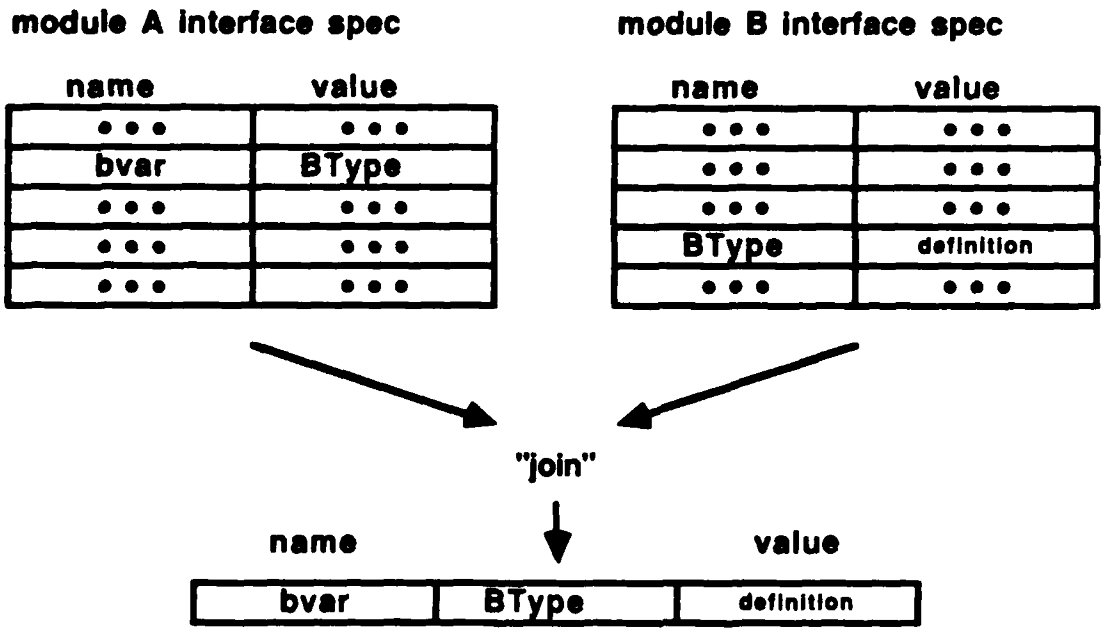


Figure 7: Simplistic relational tableau

example, encompassing multiple interface specification files and multiple associations (such as might be induced by overloaded function names). Several other liberties with the relational model have been taken to simplify the presentation here; a much more complex formal model is actually required to properly express this as relational operations. In addition, the complex data structures used in compilers do not lend themselves easily to the tableau representation common in relational models; but the semantic problems are quite similar. By thinking of the problem in terms of "surrogate keys" [5] a model closer to the non-database world of "pointers" can be created. I now believe no extensions to the IDL language are required to support this mechanism; rather, a higher-level IDL-based "applications" tool would provide the necessary functionality, providing the mechanisms and support code by which such complex interfaces can be crafted by clever code rather than clever coders.

Another relationship to database work is in the use of various normalization methods. Normalization is a class of operations on database structure which have the property of reducing what are called "update anomalies" by guaranteeing that each object of interest has but a single definition point. Normalization tends to run counter to the need for redundancy for performance reasons; in fact, a nontrivial amount of database engineering seems (to an outsider such as myself) to center around how to achieve both a normalized database and one which performs adequately. This is complicated by the fact that the higher-order normal forms have no algorithms by which they may be achieved given a completely arbitrary database structure as a start.

Consider the case of how to distribute information in a tree or graph structure. Classically, implementors have expended a tremendous amount of design effort saying things like "the cost of recomputing this value is quite high, so I'll stuff a local copy of it here in this node..." and "these two pieces of data are the same, so I'll create one instance and share it by creating a new node type and a pointer from the other places...". Many of these problems have already been addressed in the database world. In particular, new work on distributed databases has to deal with the high cost of not caching certain computations on the local machine. Classical normalization deals with sharing common data and preventing update anomalies. Tremendous amounts of effort go into debugging problems in compilers and other classical system structures caused by update anomalies, errors in sharing, failures to share, dangling pointers, etc. and these appear to me to be the result of the same *ad hoc* engineering decisions that plagued the database world prior to the introduction of automatic and semi-automatic design aids and sound theoretical bases for describing data.

Many of the design tradeoffs made in classical data structure engineering (such as replication of data, additional pointers, etc.) deal with performance issues, and knowledge of the semantic domain which indicates which information is likely to or will remain constant. In the database world, it appears that everything can change, and because of the long-lived data, almost certainly *will* change given enough time. In system structures, the data at some point in the processing is frequently read-only (and in linear systems such as compilers is never modified once it has been created). Thus current techniques are not directly applicable. Nonetheless, some automated help on the design of data structures is certainly desirable. Having such automated tools work on IDL-level descriptions which are both language- and machine-independent strikes me as more valuable than tools which would work on Ada, Pascal, C, or other language-specific representations. Since the lower-level language specific representations can be derived from the IDL specifications, there is a significant gain in applicability by using the IDL level model.

The result is that by having high-level specifications of the data, and in particular by the use of specifications that allow an automated system to deduce usage (e.g., write-once vs. read/write, or read-only after a certain point in the lifetime) extensions of many of the normalization techniques used in the database domain will find applicability to IDL definitions. Some of the tasks currently performed by those writing IDL definition files, such as creation of classes (especially non-hierarchical classes), creation of views, and the choice of how attributes are assigned to various nodes and classes may well become automated. In addition, because of the richer information on lifetimes new kinds of "normalization" not applicable in the general database domain may become apparent. This is an interesting area for future research.

10 Object Based Systems

One issue raised at an SEI workshop session on future technology was the issue of active databases and object-based systems. As the balance shifts from long-lived programs and short-lived data to long-lived data and short-lived programs, problems such as semantic consistency are creeping in. If the database is extended to support program X, and program Y (which has been around for years) does not understand how to maintain the invariants required by program X, then any use of program Y on the database may destroy its integrity as far as program X's view. This is the classical "editing through views" problem [9].

The core of this problem is that the semantics are embodied not in the data but in the programs which manipulate it. Active databases, particularly those based on the object-based paradigm, appear to be a way of achieving the necessary consistency.

IDL includes an "assertion language" in which invariants about the data structure may be written. Using a transaction-based model for updating a database, such that the invariants are expected to hold after the completion of each transaction, it should be possible to validate the integrity of an IDL database after each transaction. Using specification systems such as attribute grammars based on IDL [16], it should be possible to create databases which maintain their consistency. These appear to be interesting areas for research.

11 Non-traditional Data Presentation

The use of graphical input languages has increased dramatically in the last few years. However, most of these languages deal with the high-level design aspects. Some create COBOL or PL/1 data definitions for the database design. However, in most of these systems there is no inverse operation; given an instance of a data object there is no way to graphically display the data during debugging. In most cases, graphic display of the data is based on sets of data objects and is restricted to pie charts, histograms, and other "business graphics". The simplifying assumption of "no pointers" also makes the display in some of these systems rather simple compared to what is needed to support an IDL based system.

While some of what is presented here seems obvious, in fact very few systems actually provide the necessary support for complex information output. Graphical output, discussed later in this section, is an old desideratum, but very few systems in everyday production use for systems programmers actually provide it (whereas in the database world it is becoming commonplace). Our experience at CMU and Tartan with a variety of graphical output systems resulted in the incorporation of several specialized graphical output systems as an integral component of the Tartan development environment, and which are in daily use by many users. Even the simple text forms for complex structures are beyond what most debuggers provide today, particularly if more than one level of structure is to be displayed.

A substantial piece of power and flexibility in the CMU LG system and the Tartan IDL implementation came from the fact that the a complete run-time symbol table was available during development. With a debugger interface this allowed the developers to display the internal data structures in the IDL external form.

A matter of considerable debate arose during the LG and IDL efforts. This dealt with the "human factors" issues of how information is presented to the user. The LG system would display its data in a "flat" form; for example, a simple addition tree for "17+22" might be displayed as

```
1: binary
   (op +)
   (left 2:)
   (right 3:)

2: const
   (value 17)

3: const
   (value 22)
```

However, for deeply nested structures the flat form made understanding of the actual structure very difficult, since the distance on the listing between nodes was often based on an NLR treewalk[8]. We spent a lot of time either rearranging the output with a text editor for easier examination or drawing lines on listings.

It was obvious from this experience that the "right" representation was a nested tree representation. Consequently, when IDL was implemented, a nested form was used:

```
L1103: binary [op plus; left
     L1747: const [value 17];
     right
     L406: const [value 22]]
```

This turns out to be at least as bad to read; the labels are completely arbitrary values (they happened to be the machine addresses of the nodes, because that was convenient when debugging, but many other naming conventions could have been used). Since there is generally no look-ahead, a label must be displayed because it *might* be referenced (in the above example, L1747 and L406 are not referenced). When references did appear (as in a dag or cyclic graph),

[8]Node, Left subtree, Right subtree. Notation due to W.A. Wulf.

locating the referenced node in a listing was nearly impossible. A slight improvement was made on the debug output by numbering the lines of the output and encoding the line number in a label, e.g., "L1747_201^" meant that the referenced label was defined on line 201. Also, it turns out that people cannot read indentation well. Putting "ruler marks" on the listing helped some; an example of such output (with a reference to a label given) is shown below.

```
 200| ....|....|....L1103: binary [op plus; left
 201|     |    |    |   L1747: const [value 17];
 202|     |    |    |   right
 203|     |    |    |   L406: const [value 22]]
...
1507|     |    |    |    |L2347: unary [op unary_minus;
1508|     |    |    |    |         operand    L1747_201^]
```

In spite of this, we ended up drawing a lot of lines on listings.

Some of the applications actually had built-in data structure printers which worked in a domain-specific fashion. The code generator components had tree-printer utilities that displayed the internal tree structure as a tree structure on the screen (but not on a listing), in a form much like this:

```
              L1103: plus
           ______/ \______
          /               \
   L1747: 17          L406: 22
```

The attribute-grammar system we built had an internal unparser so that internal structure could be immediately related to the source; in addition various internal structures, such as symbol tables, could be "unparsed" into a meaningful display. Important properties of both of these systems included depth-limited cutoff so only the relevant material was displayed.

Both of these systems were considered indispensable by their users.

What has become obvious to me is that many people, myself included, tend to *think* of data structures graphically, reduce the thoughts to strings of ASCII text, convert from that to a representation of bits, and at best we get back strings of ASCII text (unless we are so unfortunate as to be able to get back only the bits!); only in rare cases do we get anything approximating the model in which we first thought of the problem.

A project I hope to pursue, but which I also hope others will investigate, is the use of graphical input specifications for complex data structures which do not necessarily have a natural representation as relational tableaux. For my own goals, I hope to someday produce another highly integrated environment in which the normal form for data structure display on input or during debugging is graphical. Equally important, this should not be a passive output form but one with which the user may interact. A significant effort on the use of graphical output form has already produced a working system in the North Carolina SoftLab project [21].

12 Conclusion

This paper has presented, rather briefly, some experiences with IDL. These experiences suggest some possible future directions for IDL-based tooling and the IDL language itself. These include, as possible research, development, and/or engineering ideas:

- Design of target languages supporting the IDL type model,
- Providing support for separate compilation of IDL specifications for use as components in a system,
- Providing support for ameliorating some of the problems of large system construction, particularly those following from the use of IDL or other high-level descriptive languages,
- Providing graphical data presentation for both specification and debugging,
- Designing automated design support tools for the construction of complex IDL definitions.
- Designing support for persistent data objects ("database" support), or support for object-based models using IDL specifications.
- Using and adapting techniques from the database world to allow for more correct and robust data design without compromising the performance of a system.

Acknowledgements

The work described here actually began about 1977 with the PQCC project. It would be impossible to list the dozens of people who contributed to the LG and IDL work over that time. Primary and core contributors, however, include Paul N. Hilfinger and Steven O. Hobbs, major contributors to the LG design, and John R. Nestor, David Alex Lamb and William A. Wulf, designers of the IDL language. Valuable experience was gained at Tartan Laboratories on the use of IDL in a practical environment, and thanks are extended to all those from Tartan who provided valuable feedback on IDL, including those who made significant contributions to the IDL system implementation. This includes but certainly is not limited to Edward N. Dekker, David A. Syiek, Guy L. Steele, Jr., John Nestor, David Lamb, Steven B. Byrne, and Kenneth J. Butler. Continuing discussions with John Nestor, David Lamb, Richard Snodgrass, and Karen Shannon (among many others) have been quite valuable. I apologize to anyone I have inadvertently omitted. Thanks to the many reviewers of this paper, especially David Alex Lamb and David Garlan. [9] Purvis Jackson of SEI provided both critical input and significant production support.

References

[1] Technical Committee X3H4.
Information Resource Dictionary System.
Draft Proposal, American National Standards Institute, April, 1985.

[2] .
Bliss Language Guide.
Technical Report, Digital Equipment Corporation, 1978.

[3] Borison, Ellen.
A Model of Software Manufacture.
In *Proceedings of the Workshop on Advanced Programming Environments.* IFIP WG2.4, Trondheim, Norway, June, 1986.
To be published by Springer-Verlag.

[4] Shannon, K. and Snodgrass, R.
Candle: A common Attributed Notation for IDL.
March, 1986.
copyright 1986.

[5] Codd, E.F.
ACM Transactions on Database Systems.
In *Extending the Database Relational Model to Caputre More Meaning.* , 1979.

[6] Goos, G. and Wulf, W. A. (editors).
Diana Reference Manual.
Technical Report CS-81-101, Carnegie-Mellon University, Computer Science Department, March, 1981.

[7] Evans, A., Jr. and Butler, K. J. (editors).
Diana - An Intermediate Language for Ada.
Springer-Verlag, 1983.

[8] Firth, R.
private communication.

[9] Garlan, D.
Views for Tools in Integrated Environments.
In *Proceedings of the Workshop on Advanced Programming Environments.* IFIP WG2.4, Trondheim, Norway, June, 1986.

[10] Newcomer, J. M., Cattell, R. G. G., Dill, D., Hilfinger, P. N., Hobbs, S. O., Leverett, B. W., Reiner, A., Schatz, B. and Wulf, W. A.
PQCC Implementor's Handbook.
CMU Internal Technical Report, Carnegie-Mellon University, October, 1980.
copyright 1978, 1979, 1980.

[11] Evans, A., Jr., Morgan, C.R., Greenwood, J.R., Zarnstorff, M.C., Williams, G.J., Killian, E.A. and Walker, J.H.
Praxis Language Reference Manual
Lawrence Livermore Laboratory, 1981.

[12] Nestor, J. R.
Revised - Process Model for IDL.
informal presentation, IDL Implementors' Workshop, Kiawah, Island, May, 1986.

[13] Nestor, J. R., Wulf, W. A. and Lamb, D. A.
IDL - Interface Description Language - Formal Description.
Technical Report CS-81-139, Carnegie-Mellon University, Computer Science Department, August, 1981.

[14] Nestor, J. R., Wulf, W. A. and Lamb, D. A.
IDL - Interface Description Language - Formal Description.
(draft revision 2), Software Engineering Institute, March, 1986.
reprinted with permission of the authors.

[15] Thacker, C. P., Sproull, R. F. and Bates, R. D.
SIL,Analyze, Gobble, Build: Reference Manual
1981.

[16] Nestor, J. R., Mishra, B., Scherlis, W. L. and Wulf, W. A.
Extensions to Attribute Grammars.
Technical Report TL 83-36, Tartan Laboratories Incorporated, April, 1983.

[17] Tichy, W. F. and Baker, M. C.
Smart Recompilation.
In *ACM 1985 Conference Proceedings.* January, 1985.
published in Principles of Programming Languages, ACM, pp 236-244.

[18] Snodgrass, R., (editor).
IDL Manual Entries (Version 2.0)
Computer Science Department, 1985.
copyright 1985.

[19] Warren, W. B., Kickenson, J. and Snodgrass, R.
A Tutorial Introduction to Using IDL.
SoftLab document No. 1, University of North Carolina at Chapel Hill, Computer Science Department, November, 1985.

[20] Shannon, K. and Snodgrass, R.
Mapping the Interface Description Language Type Model into C - Extended Summary.
Internal Document, University of North Carolina at Chapel Hill, Computer Science Department, 1985.

[21] Butler, N., Curry, J., Konstant, S. and Rosenblumm, D.
Treepr Users Manual
SoftLab document No. 4 (copyright 1985) edition, 1985.